Dynamic Analysis of Multithreaded Embedded Software to Expose Atomicity Violations

by

Jay Patel

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved March 2016 by the
Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Fengbo Ren
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

Concurrency bugs are one of the most notorious software bugs and are very difficult to manifest. Significant work has been done on detection of atomicity violations bugs for high performance systems but there is not much work related to detect these bugs for embedded systems. Although criteria to claim existence of bugs remains same, approach changes a bit for embedded systems. The main focus of this research is to develop a systemic methodology to address the issue from embedded systems perspective. A framework is developed which predicts the access interleaving patterns that may violate atomicity using memory references of shared variables and provides support to force and analyze these schedules for any output change, system fault or change in execution path.

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Yann Hang Lee for being patient with me and guiding me through important transition of my career from hardware to software.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Concurrency bugs in multi-threaded software are among the most difficult to handle. They are difficult to manifest as they require specific interleaving to get triggered. The consequences of this kind of bugs are events like Northeast Blackout and NASDAQ glitch. Reliability in real time embedded systems is the most important thing. A glitch in auto-pilot system is not affordable. Programmer may fail to foresee any particular interleaving due to huge number of possible thread interleaving. The unexpected system resets of Mars pathfinder were result of failure to notice one such possible interleaving while testing. Hence, proper testing methodology for manifestation of concurrency bugs in multi-threaded software becomes one of the most important aspect of development process. The stress testing for a large multi-threaded program may take days to execute application with all possible interleaving patterns. Hence, fast testing method which can predict the access interleaving pattern that can manifest a particular concurrency bug using the properties of that bug is necessary.

Serializability or Atomicity guarantees non-interference from other threads while executing a block of code. This means other threads cannot access or change shared data while a single thread is inside atomic block working on shared data. A programmer uses synchronization locks to make a block of code atomic. But sometimes due to improper use of locks or due to abstraction programmer does not get expected atomicity. As shown in Figure 1(a), the programmer intends to read available data from buffer on the basis of Available_Buffer_Size. Read and write of Available_Buffer_Size variable should have been atomic. If programmer uses locks as shown in figure 1(a) and (b), then interleaving shown in figure 1(a) will execute properly, but the interleaving shown in figure 1(b) is a clear atomicity violation as value of Read_Length used by Read_Buffer_Data() in thread 1 is a stale value of Available_Buffer_Size. These type of programming errors may lead to program crash, invalid output generation or change in execution flow of thread. Concurrency bugs characteristic study [1] shows atomicity violation bugs are one of most common bugs apart from data race and

deadlock bugs. The example shown in Figure 1 is very simple but due to abstraction of functions or methods it really becomes difficult for programmer to guess.

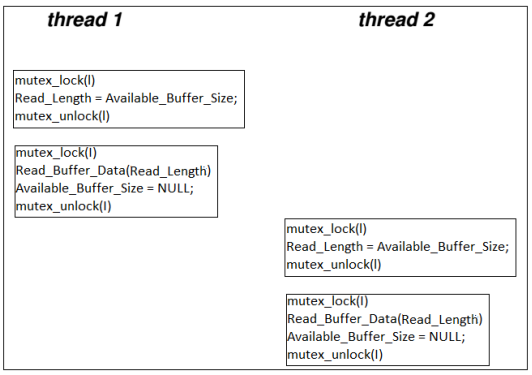


Figure 1(a): Code example with interleaving giving no error

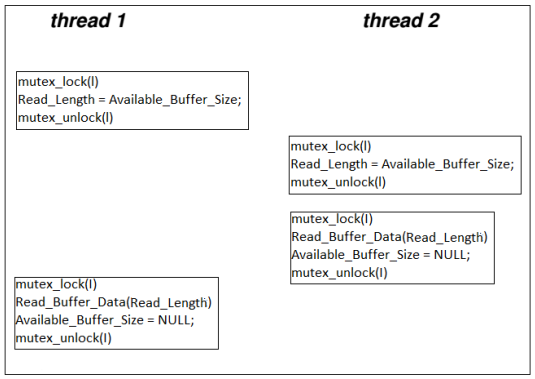


Figure 1(b): Code example with interleaving giving error

A framework is constructed using systematic approach to expose atomicity violation bugs in software for embedded systems. Following are the contributions made by this approach.

1) Although, there are many framework available to expose these bugs, the framework developed tries to address issue from embedded system's perspective. Execution replay mechanism is adopted which records synchronization events with minimum perturbation during test run of application and then use profiling tool to record memory references in replayed run.
2) The framework provides heuristics based on code semantics using which programmer can filter out significant number of predicted access interleaving patterns that can violate atomicity.
3) The framework provides a forced execution of each predicted access interleaving pattern which enforces the pattern. This is helpful in analyzing the impact of atomicity violation in terms of output or execution path change.

CHAPTER 2

RELATED WORK

CHESS [8] provides an efficient stress testing for a multi-threaded software. The framework records synchronization events using wrappers and generates a Happen-Before graph. On the basis of this graph all possible interleaving patterns are tested. The number of possible interleaving is still relatively very high and takes huge amount of time to cover all cases.

AtomFuzzer [11] checks two consecutive access of shared variables for specific access pattern of shared variable in same thread and halts thread execution before second access to check if any other thread accesses the shared variable. The method is simple and efficient but bug is only detected when it manifests for a particular interleaving during test runs.

Another run time bug detection mechanism is explained by Ruirui, Erik and Edward in [10]. The work introduces concept of Order sensitive critical sections which is defined as pair of critical section that can lead to non-deterministic shared memory state depending on the order in which they execute [10]. It also keeps track of restrictions from all synchronization operations by checking if there is any direct or indirect strong ordering between two critical sections. For e.g. if there is barrier event between two critical section then one of the critical section has to happen before another.

Recently, good work has been done to expose violation bugs using predictive approach. In this approach, schedules are predicted using memory references of shared variable and scheduling patterns of threads during test run. Liqiang and Scott in their work [12] explains different methods which uses predictive approach to detect atomicity violations. It presents two types of methods. The first one uses Lipton's reduction theory and the other one uses a Block based algorithm. Atomizer [9] predicts atomicity violations using Lipton's theory of reduction. The block based algorithm determines whether atomicity violation is possible in memory trace obtained in observed run by permuting the order of events consistent with the synchronization events. AVIO [4], SVD [5], Atom Traccker [6], CTrigger [3], PENELOPE [2] and approach taken in this work use block based algorithm approach to predict access interleaving patterns that can violate atomicity. The work can be further classified into two groups.

The first group infers all the information related to interleaving patterns and shared variable automatically from correct test runs and on the basis of this information schedules are predicted. AVIO [4], SVD [5] and AtomTracker [6] are the frameworks developed using this approach. These techniques infers atomic regions in threads on the basis of memory access and learns access interleaving pattern that programmer expects. The approach does not require any annotation by programmer. But there are few things to consider. The first one is classification of test run as "correct" run. The prediction algorithm is trained on the basis of test runs. As a result, there may be a huge number of false positives. Although, the process of inferring information is done offline, the prediction of schedules is more complex and computationally intensive. These techniques also constraints the atomic regions by certain factors like number of variable that are accessed and number instructions that they can execute per atomic region.

The second group requires programmers to annotate information like synchronization events and shared variables. PENELOPE [2], CTrigger [3] and research in this thesis falls in this category. In this approach, events and references are observed in a single test run. Based on memory references of shared variables and events like barrier, semaphores, mutex and thread/create join, possible interleaving patterns that can violate atomicity are predicted. The difference among these processes is the efficiency and implementation of prediction algorithm.

PENELOPE [2] generates schedules using locksets and acquisition histories. But the assumption of the algorithm that threads uses only locks to interact limits its practical use. CTrigger [3] stepwise generates a list of unserializable interleaving patterns. In first step, it generates a list of all interleaving patterns that can violate atomicity. In subsequent steps, the interleaving patterns are filtered to possible interleaving patterns using the ordering of synchronization events. Moreover, it can replay each pattern.

The false positives are generated by methods of both groups because the intention of programmer cannot be predicted accurately. Hence, this approach gives programmer a set of interleaving patterns which framework thinks can violate atomicity. But the number of interleaving patterns is still relatively higher and it becomes difficult for programmer to go through each pattern. There may be some true atomicity violations in the program but it may produce nothing that concerns

programmer. ConMem [7] further filters the interleaving patterns on the basis severity of bug. It gives the patterns which can crash the program. CTrigger [3] can replay all interleaving pattern but does not provide enough support for analysis of these patterns.

When we look from embedded system's perspective it is very important to have an unperturbed execution to record correct behavior of program. Hence, a recorder with minimum overhead is a prime necessity for recording events. Binary Instrumentation for collecting trace during test runs and has significant overhead and may not give unperturbed execution of embedded software. Due to uncertainty of events in embedded systems, prediction on the basis of some test runs is not accurate.

Chess [8] and Replay Debugger [15] uses wrapper methodology to record synchronization events and the later one even records inputs with minimum overhead. Both frameworks generates happen before relationship graph of events. Replay Debugger [15] also shows that both parallel program execution can be replayed on the basis of happen before relationship of synchronization events. A systemic approach to address the issue using execution replay of embedded software is very well explained by Yong Song in [16]. The same approach has been taken to expose atomicity violations in multithreaded embedded software.

Along with prediction and enforcement of access interleaving patterns, the framework also compares execution path of different runs with original run. PIN Play [21] framework provides record and deterministic replay of multi-threaded software along with record of execution trace. The mechanism to reduce trace size using branch predictors is very well discussed in [19], [20] and [21].

CHAPTER 3

ATOMICITY VIOLATION BUGS

Atomicity or Serializability, is a property for several concurrently executed actions, when their data manipulation effect is equivalent to that of serial execution of them [4]. There are many cases as discussed in earlier section where programmer expects atomicity but due to improper use of locks or too much of abstraction, the expected atomicity is not maintained. It should be noted that there is no data race (i.e. accesses to shared variable are atomic) but consecutive accesses that should have been in same atomic region or should have been made atomic using same lock event causes atomicity violation. In most of the cases, abstraction is the culprit in which programmer uses two different functions in same thread which accesses a shared variable using internal locks not known to programmer due to abstraction. As a result, there is a possibility of some remote thread accessing the variable in between these two functions. In this case, either the remote thread reads an intermediate value or corrupts the value by writing it. Hence, this may result into a different output or system faults even if there is no data race, no change in inputs or execution path.

Analysis of access interleaving patterns of two threads and code semantics are useful to detect atomicity violations. The strategy is simple. If we define two consecutive access to a shared variable by same thread as P(previous) and C(current) and access to same shared variable by a remote thread as R(remote), then access order {P, R, C} (P and C are interleaved by R) can be called a atomicity violation. Not all access patterns can be classified as atomicity violations for e.g. access pattern {P: Read, R: Read, C: Read} because the result will be same even if the access pattern is {P, C, R}. Hence, on the basis of this logic, four unserializable access patterns out of eight possible access patterns can be classified as probable atomicity violations. These access patterns are shown in figure 2.

Each patterns are justified with examples below:

1) {P: Read, R: Write, C: Read} - Two reads will have different value of same shared variable due to intermediate update by remote thread. The real world bug can be found in Apache server code shown in figure 3(a)
2) {P: Write, R: Write, C: Read} - The local read gets a corrupted value due to intermediate

write by a remote thread. The real world bug can be found in Mozilla code shown in figure 3(c).

3) {P: Read, R: Write, C: Write} - The local write intending to update the value read in previous access updates a dirty value. This is the most common atomicity violation access pattern. Even example shown in the introduction in figure 1 is a violation due this type of access interleaving pattern.

4) {P: Write, R: Read, C: Write} - Remote thread read an intermediate result. The real world bug can be found in MYSQL code in figure 3(b).

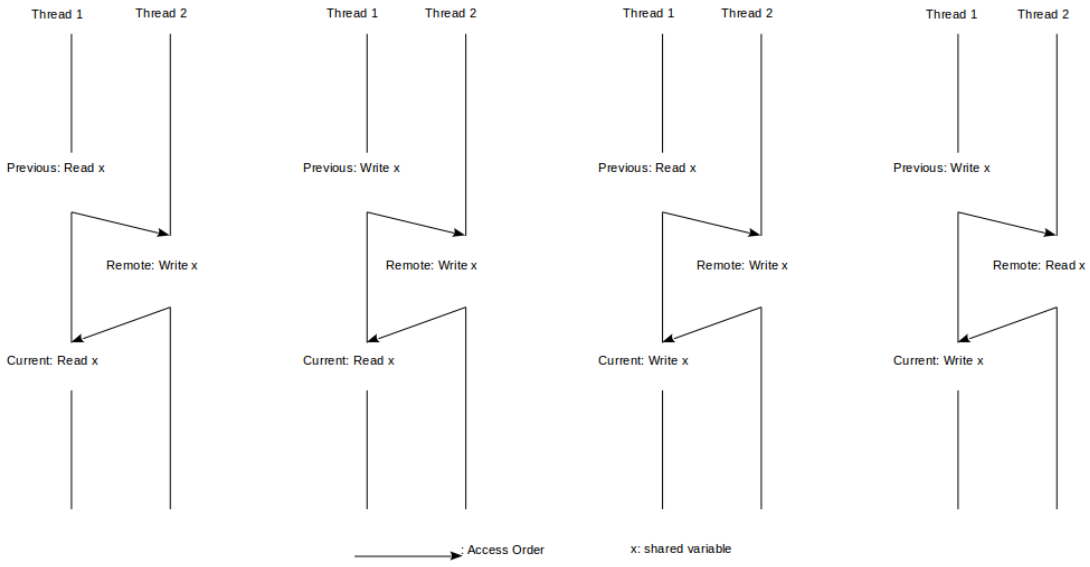


Figure 2: Unserializable access interleaving patterns

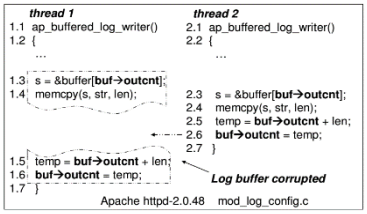


Figure 3(a): Violation bug in apache server code

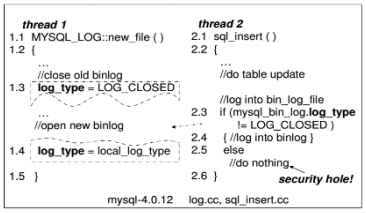


Figure 3(b): Violation bug in MYSQL code

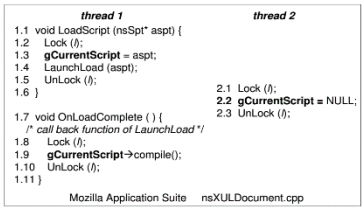


Figure 3(c): Violation bug in Mozilla code

The classification by the above scheme will give all possible cases with many false positives, as intention of programmer to access a variable consecutively in same thread cannot be predicted accurately. The list can be further filtered by looking at code semantics. One of the important thing that should be considered is the distance between two consecutive accesses. This is based on simple assumption that if programmer expects atomicity in two consecutive accesses then there should not be more instructions executed between two accesses. The distance of remote access is also useful parameter to further filter out more patterns. Hence, making probability of atomicity violating access pattern extremely low. This filter will work for high performance system but will not work for embedded systems given randomness of external events which may make any interleaving possible. Other semantics includes identifying access of a shared variable in a loop by local thread. In this case, programmer access same atomic region twice but this may get reported as bug.

All these optimizations can be easily applied once all feasible access patterns are available. But the process of obtaining feasible access pattern is easier said than done and this is where happen before relationship analysis enters into the picture. Suppose all shared variables are statically defined then it is possible to come up with list of access interleaving patterns that may violate atomicity. But it is very difficult to come up with “feasible” patterns because it is very difficult to predict order of synchronization events statically. The classification of interleaving pattern as feasible becomes very easy with happen before relationship graph. Hence, a single execution recording synchronization events dynamically to build happen before relationship graph and memory accesses to find accesses to shared variable makes things very simple and fast. The next section covers dynamic analysis of multi-threaded software.

CHAPTER 4

DYNAMIC ANALYSIS OF MULTI-THREADED EMBEDDED SOFTWARE

Dynamic debugging especially debugging on the basis of execution trace has become vital part of software debugging process. A single threaded program can be easily analyzed and replayed by maintaining same total order of events in the program i.e. the executed instructions. The only thing that can change execution of single threaded programs are branch instructions. Hence, program can be easily replayed by recording Program Counter (instructions) and inputs to the program. In this way developer is able to see what actually happened. Many important things like memory references, stack pointers and return pointers can be recorded in the trace and analyzed later in the replay. The size of trace generated is very high and there are various optimizations that can be done to decrease the trace size.

Partial order of events comes into picture in a multi-threaded software. The partial order of events in multi-threaded program is order in which events of different threads interacts among the each other using synchronization objects. Partial order can be determined by recording events used by threads for interaction. The partial order of multi-threaded program depends on events like creation of new threads, inter-process communication among threads for signaling or to access a shared variable. Hence, to replay a multi-threaded software one needs to maintain same total and partial order of events.

Along with large amount of data generated, the performance penalty for recording trace for total order of events in execution of each thread is very high as one needs to stop current execution at every instruction (depending on granularity), record the trace and then continue execution. The performance penalty of recording partial order events depends on the methodology. If events are recorded using binary instrumentation (by using symbols for event calls) then the penalty is significantly high while the penalty is very low if events are recorded using wrappers of event functions. Hence, wrapper methodology is better to record partial order of events.

Unperturbed execution is a must requirement for embedded software. Hence, recording partial order and total order in single run is not a good idea. As shown in figure below, the process of recording trace can be divided in to two parts by first recording partial order and then enforcing

partial order to record total order. The total order will only change if there are different inputs or a different partial order. Hence, the original execution can be replayed by maintaining same inputs and partial order. The trace can be recorded in this replay run.

The figure below from [16] summarizes the systematic methodology to dynamically analyze multi-threaded software using execution replay.

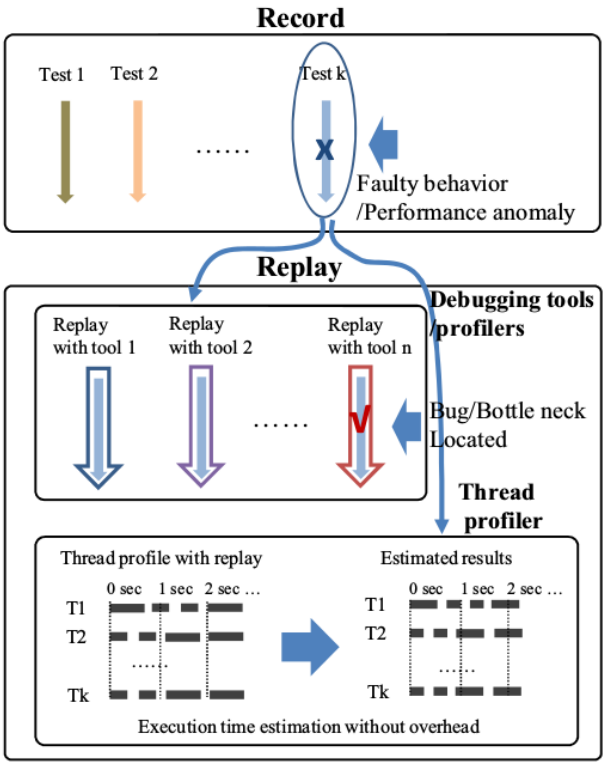


Figure 4: Dynamic Analysis with execution replay. Figure taken from [16]

CHAPTER 5

FRAMEWORK

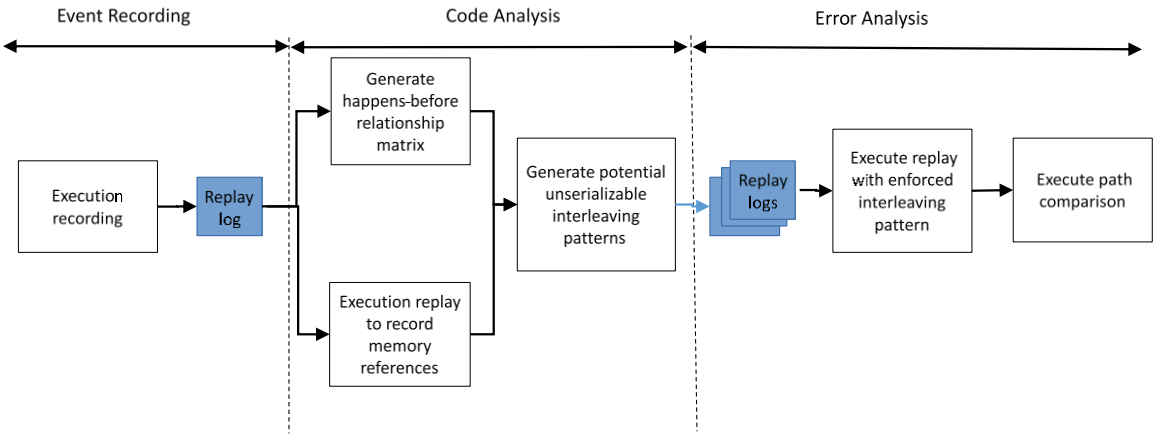
A hybrid framework using both wrapper methodology and binary instrumentation is developed to expose atomicity violation bugs. The process of recording information is divided into two parts. A wrapper methodology ensures recording of partial order events among threads with minimum perturbation and other execution information like memory accesses to shared variables are recorded using binary instrumentation.

The framework shown in figure 5 consists of three main phases as follow.

1) Event Recording: A low overhead recorder records synchronization and IO events and generates a happen before relationship graph of events. POSIX libraries APIs are wrapped using custom function keeping the interface same. Hence, user can use recorder without making much change to program.
2) Code Analysis: The program is replayed on the basis of happen before relationship graph generated in previous phase. The replay of program is binary instrumented using developed Intel PIN tool to record the memory references of shared variables in critical sections. A binary happen before relationship matrix is also generated using graph which shows if there is any happen before relationship between any two events in graph. Order generator generates all feasible access interleaving pattern which may violate atomicity using memory references to shared variable by different threads and happen before relationship matrix.
3) Error Analysis: The application is executed for different execution orders under a controlled environment in which each access interleaving pattern generated in phase 2 is enforced. The execution trace of each run is collected using binary instrumentation to analyze whether there is any change in execution path, output or system fault due to possible violation. A branch prediction mechanism is used to collect trace to reduce trace size.

Each block of framework is explained as follow. The code example shown in figure 6 will be used to explain each stage of framework.

Figure 5: Framework



Thread 1:
1: pthread_create(thread1);
2: pthread_create(thread2);
.
.
.
3: pthread_join(thread1);
4: pthread_join(thread2);

Thread 2:
1: mutex_lock(lock1);
2: Read_Variable(X);
3: mutex_unlock(lock1);
.
.

4: barrier_wait();
.
.
.
5: mutex_lock(lock1);
6: Write_Variable(X);
7: mutex_unlock(lock1);
.
.
.

8: mutex_lock(lock1);
9: Read_Variable(X);
10: mutex_unlock(lock1);

Thread 3:
1: barrier_wait();
.
.
.
2: mutex_lock(lock1);
3: Write_Variable(X);
4: mutex_unlock(lock1);
.
.
.
5: mutex_lock(lock1);
6: Read_Variable(X);
7: mutex_unlock(lock1);

Figure 6: Example Code

Record-Replay

As discussed earlier a low overhead recorder is an essential part of the framework to have an unperturbed execution of embedded software during test run. Happen before relations between events are captured by chains of immediate happen-before relations. Wrappers for POSIX events like semaphore, mutex, conditional wait and barriers substitutes the actual system calls.

The inter thread and intra thread dependencies are recorded in the log file. Each thread and synchronization object used maintains a sequence of events. The thread sequence keeps track of order of events that happens in a particular thread while synchronization object sequence keeps track of order in which events access synchronization object.

The partial order graph of one possible execution of the example code in figure 6 is shown in figure 7. The red colored digits shows the global sequence of events. Sequence of events maintained by thread will capture happen before relationship of events in same thread. For e.g. Thread 1 and Thread 2 will maintain following sequence.

Thread1_Sequence = {1, 5, 6, 7 8, 9, 10, 11} Thread2_Sequence = {3, 4, 12, 13, 14, 15}

One mutex object used also maintains a sequence of events accessing mutex. The mutex will maintain following sequence

Mutex_Sequence = {5, 6, 8, 9, 10, 11, 12, 13, 14, 15}

Each log entry maintains immediate happen before relationship. Log tuple for each event in log file is as follow.

Log = < Event_Type, Tid, Thrd_Clk, Event_Index, Thread_Dependency_Event, Event_Dependency_Event, Event_Executed, File_Name, Line_Number>

Where **Event_Type** is type of synchronization event for e.g. mutex_lock, sem_wait() etc, **Tid** is thread invoking the event, **Thrd_Clk** is the sequence number of event in the same thread, **Event index** is id of event of a specific type for e.g. if two mutex are used then **Event_Index** will be different for both of them (Hence, {**Event_Type, Event_Index**} gives the particular synchronization and communication object used by threads to interact among each other), **Thread_Dependency_Event** is the event in the same thread to which event is dependent, **Event_Dependency_Event** is the event with same **Event_Index** to which event is dependent,

Event_Executed shows whether event is executed, **File_Name** is the name of code file in which event is coded, **Line_Number** is line in code file where event is coded.

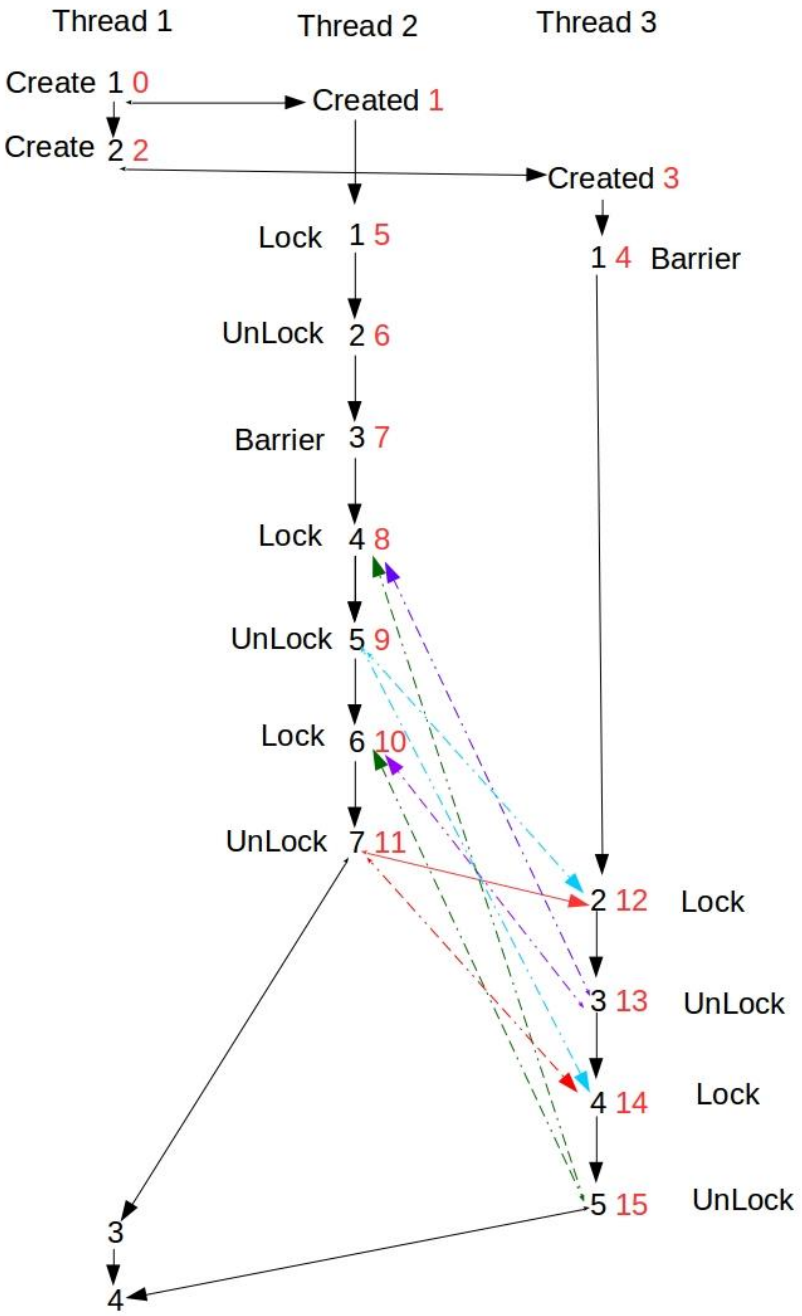


Figure 7: Partial order graph for example shown in figure 6 created one possible execution

Table 1 shows the log file generated by recorder for example code given in figure 6. In the example given in figure 6, there are three threads. There are three types of events: thread create, barrier and mutex lock. There in total four synchronization objects: two thread create, one barrier and one mutex. It can be noticed that event index corresponding to each synchronization object is unique for a particular event type. Negative index is used to differentiate opposite actions. For e.g. Event index mutex_lock of mutex synchronization object is '1' while mutex_unlock of same mutex synchronization object is '-1'. Thread Dependency and Event Dependency are the fields that captures immediate happen before relationships. Mutex event in thread 2 with global event ID 5 depends on event 1 (thread created). Mutex event in thread 2 with Global event ID 8 has event 7 as thread dependency and event 6 which mutex_unlock event as event dependency. Similarly Mutex_event in thread 3 with Global ID 12 has event 4 as thread dependency and event 11 which mutex_unlock event in thread 2 as event dependency.

Global Event ID	Event Type	Tid	Thread Clk	Event Index	Thread Dependency	Event Dependency	Executed	Filename	Line Number
0	0	1	1	-1	-1	-1	1	Example.c	71
1	0	2	2	1	-1	0	1	Example.c	71
2	0	1	2	-2	0	0	1	Example.c	71
3	0	3	3	2	-1	2	1	Example.c	71
4	12	3	4	1	3	4	1	Example.c	0
5	3	2	3	1	1	-1	1	Example.c	37
6	3	2	4	-1	5	5	1	Example.c	39
7	12	2	5	1	6	7	1	Example.c	0
8	3	2	6	1	7	6	1	Example.c	44
9	3	2	7	-1	8	8	1	Example.c	46
10	3	2	8	1	9	9	1	Example.c	48
11	3	2	9	-1	10	10	1	Example.c	50
12	3	3	10	1	4	11	1	Example.c	44
13	3	3	11	-1	12	12	1	Example.c	46
14	3	3	12	1	13	13	1	Example.c	48
15	3	3	13	-1	14	14	1	Example.c	50

Table 1: Log file generated by recorder

The replay of the program is done by enforcing the partial order shown in figure 7. Before executing any event, scheduler is invoked. The scheduler makes sure that all events that happen before current event are executed. This is done by traversing the graph backwards and executing all events that happened before the current event. After making sure all events are executed, scheduler will allow thread to continue. All events that happens before current event must be executed. This is done traversing the graph backwards and executing all events that happened before the current event.

Happen-before analysis

Partial order graph is acyclic unidirectional graph having edge directed towards the node that happens latter. The partial order of events can be divided into strong and weak order for a single execution.

The strong order is the bold black edge in figure 7 which needs to happen before in every possible scheduling pattern. For e.g. the barrier event (1) in thread 3 needs to happen before lock event (5) in thread 1 in every execution. Barrier, thread create/join and semaphore are taken as strong ordering events. It is assumed that semaphore is only used for signaling.

The weak order is the bold red edge. It is one “possible” edge which may not be seen in every execution. The edge depends on scheduling of threads. The colored dashed edges shows the other possible edges that a partial order graph can have. Hence, events can be divided into strong ordering events and weak ordering events. Mutex is taken as weak ordering event.

Adjacency matrix can be built using this graph. In constructing matrix the edges with strong order are only considered. Adjacency matrix for the example shown in figure 6 shown in figure 8(a).

The Adjacency matrix is used to build Check_Happen_Before (event1, event2) function which returns happen before relationship between any two elements. The pseudo code for the function is shown in figure 9.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8(a): Adjacency matrix created for example shown in figure 6

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1
3	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1
4	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
7	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8(b): Happen before matrix created for example shown in figure 6

If event 2 node can be reached from event 1 node or vice versa then there is happen before relationship between two events. The nodes are parallel if both nodes are not connected. The reachability from one event to another can be found using depth-first search. A happen-before relationship matrix as shown in figure 8(b) can be constructed using adjacency matrix during initialization of analysis program. Once matrix is computed, then computational complexity of checking happen before relationship in the later stage of program is O(1). However, complexity of constructing matrix is significantly high and memory complexity is $O(n^2)$, where n is number of events in partial order graph.

```
Check_Happen_Before(event1, event2){
 If(Depth_First_Search(event1, event2))
     return event1->event2
 else If(Depth_First_Search(event2, event21))
     return event2->event2
 else return event1||event2
}
```

Figure 9: Pseudo code for Check Happen Before function

Apart from memory and computational complexity, one of the major drawback is that user defined synchronization primitives will not be considered in happen before relationship. For e.g. as shown in figure 10 below, Splash 2 benchmarks implement barrier using pthread_conditional_wait() instead of directly using pthread_barrier_wait(). Hence, happen before analysis will fail to recognize the implementation as barrier.

```
define(BARRIER, `{
        unsigned long  Error, Cycle;
        int            Cancel, Temp;
        Error = pthread_mutex_lock(&($1).mutex);
        if (Error != 0) {
                printf("Error while trying to get lock in barrier.\n");
                exit(-1);
        }
        Cycle = ($1).cycle;
        if (++($1).counter != ($2)) {
                pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &Cancel);
                while (Cycle == ($1).cycle) {
                        Error = pthread_cond_wait(&($1).cv, &($1).mutex);
                        if (Error != 0) {
                                break;
                        }}
                pthread_setcancelstate(Cancel, &Temp);
        } else {
                ($1).cycle = !($1).cycle;
                ($1).counter = 0;
                Error = pthread_cond_broadcast(&($1).cv);
        }
        pthread_mutex_unlock(&($1).mutex);
}')
```

Figure 10: User defined Barrier implementation for Splash 2 Benchmarks

Binary Instrumentation

Intel PIN framework has been used for binary instrumentation. PIN is a dynamic binary instrumentation framework that enables creation of dynamic program analysis tools. The framework allows to build custom profiling tools called PIN tools using available PIN APIs. Instrumentation is done at runtime using just-in-time compiler. Thus, it requires no recompiling of source code. It also provides support for multithreading.

A PIN tool has been created which records memory reference to shared variables by each thread in the replay run of application based on happen before relationship captured by recorder. It records instruction pointer, memory location and operation for a particular reference. It is assumed that there is no data-race in the program. Hence, any access to shared variable is done using locks. Along with reference information, the information about the lock event using which the shared variable is referred is also recorded. The information like node number of event in partial order graph and event to which lock event is nested (if nested) is recorded. The tool also looks for functions symbols inserted in Replay library to specify range of recording. In this way, only the code executed in user application is recorded. For e.g. the memory access done in provided wrappers of replay library are not recorded. The memory references of user application are only recorded.

Each reference in log is represented by the following tuple.

Reference = <Operation, Instruction, Address, Thread_ID, Event Node, Nested_to>

where **Operation** is read/write operation, **Instruction** is instruction pointer, **Address** is the memory location of variable, **Thread_ID** is thread accessing the variable, event node is the number of lock event in partial order graph, **Nested_to** is the event to which lock event is nested (if nested).

PIN tool creates a new bucket whenever a new thread is created. All references to shared variable by a particular thread are stored in thread specific bucket.

The thread buckets created for example shown in figure 6 are shown below in figure 12. As shown in buckets there are three references made by thread 2 and two references made by thread 3. The Memory address, instruction pointer and lock event node number for corresponding lock event number can also be noticed in tuple for each reference.

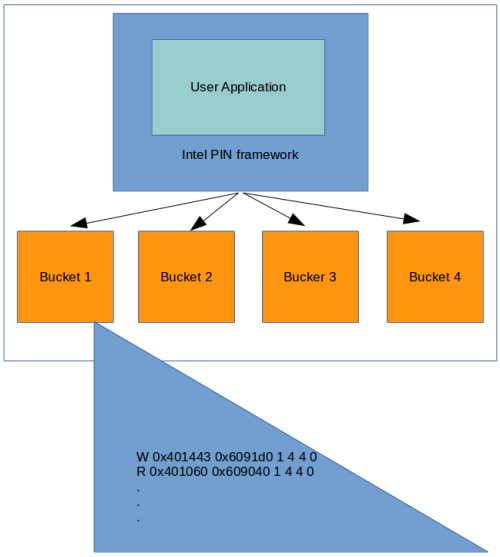


Figure 11: Overview of Binary Instrumentation tool

```
Thread 1 Bucket
R 0x4013da 0x6091cc 2 5 0
W 0x401428 0x6091cc 2 8 0
R 0x401456 0x6091cc 2 10 0
```

```
Thread 2 Bucket
W 0x401428 0x6091cc 3 12 0
R 0x401456 0x6091cc 3 14 0
```

Figure 12: Thread buckets created for example in figure 6

Order Generation

The order generator uses happen before analysis in section 4.2 and thread buckets containing Memory references to generate access interleaving patterns that can violate atomicity. It looks for two consecutive memory reference by a thread in corresponding thread bucket. Once Previous and Current events are recognized for a particular reference, the generator looks for Remote event or memory reference for the same variable in other thread buckets. After getting {P, R, C} events, the generator checks whether the access interleaving is among the unserializable pattern and then checks the happen before relationships of {P, R} and {R, C}. If R does not happen before P, and C does not happen before R, then that particular interleaving is potential violation. The algorithm for order generator is described in figure 14. The results given by order generator for the example shown in figure 6 are shown in figure 13. Events in three event columns correspond to the events numbers in partial order graph shown in figure 7. The access inter-leavings predicted are events {5, 12, 8}, {8, 12, 10} and {12, 8, 14}. {8, 12, 10} and {12, 8, 14} are clear violations and are correctly predicted. But {5, 8, 12} is predicted because it satisfies {R, W, R} criteria and event 8 is parallel to event 12. This is a false positive.

Address	Thread Id	Previous	Operation	Event	Current	Operation	Event	Remote_Thead	Reomte	Operation	Event
6091cc	2	P	R	5	C	R	8	3	R	W	12
6091cc	2	P	W	8	C	R	10	3	R	W	12
6091cc	3	P	W	12	C	R	14	2	R	W	8

Figure 13: Output given by Order Generator for example shown in figure 6

```
Order_Generator(){
 For(bucket = 0; bucket < Total_Buckets; bucket++){
  Get_P_C_events(i);
  For( I = 0; I < Total_P_C_events; I++){
   For(bucketR = 0; bucketR < Total_Buckets; bucketR++){
    if(bucketR != bucket){
     Get_R_events(bucketR, Memory_Address_P_C);
     For( J = 0; J < Total_R_events; J++){
     Check_access_interleavving criteria(P, R(J), P)
     check_happen_before(R(J), P)
      If(R(J) -> P) break;
      check_happen_before(C, R(J))
      If(C -> R(J)) break;
      Print {P, R(J), C}
}}}}}}
```

Figure 14: Pseudo code for Order generator

Analysis

The order generation part gives all possible access interleaving patterns that are potential atomicity violation. But there will be many false positives as programmer's intention cannot be predicted accurately. The analysis part can be divided in to two parts. The first one is based on code semantics and second is based on execution with forced interleaving.

The result shown in figure 13 gives good information about violations but does not give information about the location of present, current and remote events in the code, instruction pointers of these instructions, number of instructions in between present and current events. The analysis part provides heuristics using which the programmer can further filter the list given by order generator. These heuristics are based on the code semantics.

The information like instruction pointer, line number in code and number instructions between present and current event are already recorded during binary instrumentation run and happen before analysis. Hence, giving access of this information to programmer along with violations can be significantly helpful to programmer. Following are some filters which a programmer can use this information.

Loop: If instruction pointer for Previous and Current event are same then it can be referred as a loop.

Local Distance: If there is huge number instruction in between Previous and Current event then there is high probability that they are not atomic.

Code Debug: Using line number at which event is coded in the code, programmer can manually go and analyze events.

One more optimization can also be applied. If user declares the test run as correct run then the access interleaving patterns that already exist in test run can be ignored. This is one step closer to the intension of programmer.

In the second part of analysis, the access interleaving pattern is forced to analyze impact of violation in terms of output change, change in execution path and system crash. The user can either use final output or change in execution path to determine impact of violation. The shared variable affected by atomicity violation may affect internal state of the system or may change the execution

path of directly or indirectly related threads. In some embedded systems, where outputs may vary continuously, instead of directly looking at outputs the operations that generate output matters. For e.g. suppose invalid shared state of shared variable is used to configure operation parameters for some function. In this case, initial outputs generated may be same for both type of operations in a time bound testing and may not reflect wrong output while testing the system. Hence, recording execution path is one of the useful metric to measure impact of atomicity violation for embedded systems. However, it cannot be said that if there is no change in execution path then the output is correct. There can be a case where execution path is same but output is different. It depends on programmer's perspective and priority.

The access pattern can be changed by changing order of mutex acquired in happen before relationship. This is not as simple as to just change event dependency. The happen before relationship needs to restructured in such a way that it is feasible and enforces the access pattern. There are two cases described below which we need to take care of.

Case 1: Remote event happens before Previous event (weak order)

The process can be imagined as stopping the remote thread before executing Remote event till Previous event in local thread is executed and making sure that remote events grabs the lock before current event. To stop remote thread till Previous event is executed, all the nodes from Remote event which has happen before relationship {R -> nodes(i)} needs to be detached from the graph and remaining nodes in the graph should be connected till Previous event. The Remote event should be dependent on unlock event of Previous event and the next lock event should be dependent on Remote event. All nodes which were disconnected should be inserted between Previous event and later events.

Case 2: Current event happens before Remote event (weak order)

Similar process can be repeated for case two but in this case Current Thread is stopped and it waits for remote event to get executed.

In both cases, once remote event is executed, all threads are set free. No particular order is maintained. This is done to analyze the behavior of program after enforcing the interleaving.

Execution Path Analysis

The execution path of a sequential program is the order of execution of instructions which can be also defined as total order of the program. If total order of two different executions is same then it can be said that both executions has same execution path. The execution path of two different executions of same program can only be different if there are any control flow instructions in the program. Hence, if two different executions of same program, takes same branch in control flow instructions then both execution have same total order.

User application is binary instrumented using a PIN tool which records the program counter of all branch instructions executed by each thread and stores them in corresponding thread buckets. An optimization is done by using branch predictor mechanism to reduce size of trace. A branch target buffer for each branch instruction is maintained by the tool. The execution trace is only recorded if there is a miss prediction.

Two executions can be compared using the data given by PIN tool. Two separate execution traces are compared instruction by instruction. If both traces have same branch instructions taken then both execution have same total order.

Each replay of enforced access interleaving pattern in binary instrumented using the PIN tool and execution trace generated by each execution in each schedule is compared with the execution of correct test run.

CHAPTER 6

RESULTS

The testing was done on Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (8 Mb cache). Splash 2 benchmarks and some standard applications converted from Java programs are used. The following table gives a little description about all benchmarks used and synchronization primitives they use.

Test Program	Description	Synchronization Primitives
FFT	Splash 2 benchmark	Lock and Barrier
FFT with inserted bug	An atomicity violation bug is inserted while work distribution	Lock and Barrier
Lu	Splash 2 benchmark	Lock and Barrier
Barnes	Splash 2 benchmark	Lock, Barrier and Conditional
Pbzip	Compression Tool	Lock and Conditional
Banking	An example showing multiple threads trying to access one account	Lock
Load_Script	An example which loads and compile script. It replicates Mozilla atomicity violation bug	Lock

Table 2: Benchmark Description

Table 2 shows the comparison of execution time for normal run, replayed run and binary instrumented run of benchmarks to record memory trace. It can be clearly noticed that the execution time of binary instrumented code is significantly high and cannot guarantee unperturbed execution. Hence, this justifies the use of wrapper to record events instead of directly instrumenting the code.

Benchmark	Normal Execution(s)	Replayed Execution(s)	Memory Trace Record(s)
Barnes	0.05	0.05	3.885
FFT	0.03	0.03	0.84
Lu	0.184	0.194	7.493
Pbzip	0.112	0.141	3.288

Table 3: Execution time comparison for normal, replayed and binary instrumented execution

Table 3 shows the predicted atomicity violation access interleaving patterns. Actual predicted atomicity violation schedules for Barnes and Pbzip are 21 and 54 respectively. But the access interleaving patterns which were already present in test run are not considered. Also these two benchmarks does not have any actual violations. The instruction pointers of {P, C} of each interleaving are same which suggests that there is a loop. Both Barnes and Pbzip implements a FIFO queue to assign work to worker threads.

Test Program	Threads	Predicted Violating Access Interleaving Patterns	Actual Violations
FFT	4	0	0
FFT with inserted bug	4	12	1
Lu	4	0	0
Barnes	2	11	0
Pbzip	8	23	0
Banking	4	12	1
Load_Script	2	1	1

Table 4: Predicted Atomicity violation access interleaving patterns for each program

The following table shows the result of replay after enforcing the access interleaving patterns shown in table 2. When 12 access interleaving patterns are replayed after enforcing those interleaving patterns the work allocation to thread is different from original program. Hence, the execution path and output are different from original one. All interleaving patterns for PBzip and Barnes does not give any change in result of the final program or any crash. The final output of balance in banking example is different from original execution. A NULL pointer reference resulting in to segmentation fault is encountered when access interleaving is enforced for Load_Script program.

Test Program	Execution Change	Output Change	Crash	Type
FFT	-	-	-	-
FFT with inserted bug	YES	YES	NO	{R, W, R}
Lu	-	-	-	-
Barnes	-	-	-	-
Pbzip	-	-	-	-
Banking	NO	YES	NO	{R, W, W}
Load_Script	YES	-	YES	{W, R, W}

Table 5: Analysis of atomicity violation access pattern for each program

Trace compression is done while recording using branch predictor mechanism. The following table shows the comparison of trace events if number of branch instructions recorded versus number missed branch instruction recorded.

Benchmark	Branch Conditions	Missed Predictions	Compression Ratio
FFT	36885	5843	6.312681842
Barnes	12452	773	16.10866753
Lu	52997539	3077195	17.22267812
PBZip	113858618	7254590	15.6947006

Table 6: Trace Size comparison while recording Branch conditions vs Missed Predictions

CHAPTER 7

LIMITATIONS

There are various limitations of this approach. All the assumptions that are taken to build the framework act as the limitations. It is assumed that there is no data race in application which means that application should be checked with some data race detector before running through the framework. Another assumption is that user will only use built-in synchronization primitives. The tool cannot identify any user defined synchronization primitive. The user defined synchronization primitive has to be added to the framework explicitly before running the application through framework. One of the most important drawback is that the framework only detects single variable atomicity violations. It does not work for multi variable atomicity violation or correlated variables. Consider the example shown in figure 15. The update of multiple variables cache->table and cache->empty should be atomic. The framework will not detect these type of bugs.

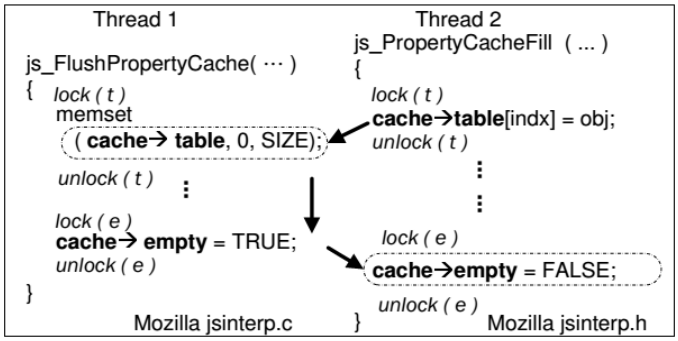


Figure 15: Multi variable atomicity violation bug in Mozilla Code. Taken from [22]

CHAPTER 8

CONCLUSION

This paper has presented a systematic approach to expose atomicity violation bugs in software for embedded systems using record-replay and binary instrumentation. The framework successfully predicts the feasible access interleaving patterns that can violate atomicity. The framework also explores the predicted access interleaving pattern by enforcing each access interleaving pattern and compares the execution trace of each enforced execution to see if there is change in execution flow due to potential violation. The framework also provides several information related to code semantics like number of instructions between two consecutive instructions, instruction pointer of memory reference and line number in code where reference is made by programmer. Using this information programmer can further prune out the predicted access interleaving patterns.

REFERENCES

1. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – A comprehensive study of real world concurrency bug characteristics. In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (Seattle, WA, USA, 2008), ACM, pp: 329-339

2. F. Sorrentinno., A. Farzan and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity Violations. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (New York, NY, USA, 2010), ACM pp: 37-46

3. S. Park, S. Lu and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2009), ACM, pp 25-36

4. S. Lu, J. Tucek, F. Qin and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (New York, NY, USA, 2006), ACM, pp 37-38

5. M. Xu, R. Bodik and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA, 2005), ACM, pp 1-14

6. Muzahid, Abdullah, Otsuki, Norimasa, Torrellas and Josep. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA, 2010), IEEE Computer Society, pp 287-297

7. W. Zhang, C. Sun and S. Lu. ConMem: Detecting Crash-Triggering Concurrency Bugs through an Effect-Oriented Approach. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (New York, NY, USA, 2010), ACM pp 179-192

8. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation (Berkeley, CA, USA, 2008), pp 267-280

9. C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA, 2004) ACM, pp 25-257

10. R. Huang, E. Halberg and G. Edward. Non-Race Concurrency Bug Detection through Order-Sensitive Critical Sections. In Proceedings of the 40th Annual International Symposium on Computer Architecture (New York, NY, USA, 2013) ACM, pp 655-666

11. C. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (New York, NY, USA, 2008) ACM, pp 135-145

12. L. Wang and S. D. Stoller. Run-time analysis to detect atomicity violations in multi-threaded program. In IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 32, NO. 2 (February 2006) pp 93-110

13. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA, 2005) ACM, pp 190-200

14. PIN 2.14 User guide

15. Y. Lee., Y. W. Song, R. Girme, S. Zaveri and Y. Chen. Replay Debugging for Multi-threaded Embedded Software. Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (Washington, DC, USA, 2010) IEEE Computer Society, pp 15-22

16. Y. Lee and Y. W. Song. Dynamic analysis of embedded software. In IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (Reno, NV, USA, 2014) IEEE, pp 166-173

17. C. Wang, R. Limaye, M. Ganai and A. Gupta. Trace-based Symbolic Analysis for Atomicity Violations. In Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems (Berlin, Heidelberg, 2010) Springler-Verlag, pp 328-342

18. Mihajlovic, Z. Zilic and W. J.Gross. Architecture aware real time compression of execution traces. In ACM Transactions on Embedded Computing Systems, Volume 14 Issue 4 (New York, NY, USA, 2015) ACM, Article No. 75

19. V. Uzelac, A. Milenkovic, M. Milenkovic and

20. M. Burtscher. Using branch predictors and variable encodings for On-the fly program training. In IEEE Transaction on Computers, Volume:63, Issue: 4 (2012) IEEE, pp 1008-1020

21. V. Uzelac, A. Milenkovic, M. Burtscher and M. Milenkovic. Real-time Unobtrusive Program execution trace compression using branch predictor events. In Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems (New York, NY, USA, 2010) ACM, pp 97-106

22. H. Patil, C. Pereira, M. Sallcup, G. Lueck and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (New York, NY, USA, 2010) ACM, pp 2-11

23. S. Lu, S. Park, C. Hu, X. Ma, W. Jhang, Z. Li, R. A. Popa and Y. Zhonu. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (New York, NY, USA, 2007) ACM, pp 1103-116

24. B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-Aware Communication graphs. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (New York, NY, USA, 2009) ACM, pp 553-563

25. A. Farzan, P.Madhusudan and F. Sorrentino. Meta-Analysis for Atomicity Violations unser Nested-Locking. In Proceedings of the 21st International Conference on Computer Aided Verification. (Berling, Heidelberg, 2009) Springer-Verlag, pp 248-262